Checking Properties of Safety Critical Specifications Using Efficient Decision Procedures*

David Y.W. Park[1], Jens U. Skakkebæk[1], Mats P.E. Heimdahl[2], Barbara J. Czerny[3], and David L. Dill[1]

[1] Computer Science Department, Stanford University, Stanford, CA 94305, USA.
[2] Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, USA.
[3] Department of Computer Science, Michigan State University, East Lansing, MI 48824, USA.
Email: {parkit,jus,dill}@cs.stanford.edu, czerny@cps.msu.edu, heimdahl@cs.umn.edu

Abstract

The increasing use of software in safety critical systems entails increasing complexity, challenging the safety of these systems. Although formal specifications of real-life systems are orders of magnitude simpler than the system implementations, they are still quite complex. It is easy to overlook problems in a specification, ultimately compromising the safety of the implementation.

Since it is error-prone and time consuming to check large specifications manually, mechanical support is needed. The challenge is to find the right combination of deductive power (i.e., how rich a logic and what theories are decided) and efficiency to complete the verification in reasonable time. In addition, it must be possible to explain why a proof fails.

As an initial approach to solving this problem, we have adapted the Stanford Validity Checker (SVC), a highly efficient, general-purpose decision procedure for quantifier-free first-order logic with linear arithmetic, to check the consistency of specifications written in Requirements State Machine Language (RSML). We have concentrated on a small but complex part of version 6.04a of the specification of the (air) Traffic alert and Collision Avoidance System (TCAS II). SVC was extended to produce a counter-example in terms of the original specification.

The efforts discovered an undesired inconsistency in the specification, which the maintainers of the specification independently discovered and subsequently fixed in the most recent version.

The case study demonstrates the practicality of uncovering problems in real-life specifications with a modest effort, by selective application of state-of-the-art formal methods and tools. The logic of SVC was sufficiently expressive for the properties that we checked, but more work is needed to extend the class of formulae that SVC decides to cover the properties found in other parts of the TCAS II specification.

*The research was supported by the Defense Advanced Research Projects Agency under contract number E276, and by National Science Foundation under NSF CAREER grant CCR-9624324.

1 Introduction

Software plays an increasingly important role in the implementation of safety critical systems. For example, avionics systems are becoming more dependent upon large amounts of embedded software to reduce cost and increase maintainability. This trend is evident in the proposed Integrated Modular Avionics (IMA) standard [15].

Unfortunately, the increase in software functionality also entails an increase in complexity, which makes assuring the safety of such systems challenging. The problem is aggravated by the fact that many safety-critical systems consist of several parallel components which are subject to real-time constraints.

Formal, high-level specifications for system software can help increase confidence in a system by revealing inconsistencies and gaps in the system design. The implementation can then be systematically reviewed for conformance with the specification. This approach has been successfully applied in several software projects [4]. Examples include the application of tables (Parnas et al.) [18] to the specification and validation of a nuclear power plant shutdown system and the application of the Requirements State Machine Language (RSML) to the specification of the (air) Traffic alert and Collision Avoidance System (TCAS II) [14].

Although a formal specification is orders of magnitude simpler than the actual system implementation, it can still be quite complex. It is easy to overlook problems in a specification that may ultimately compromise the safety of the implementation. Indeed, it has been noted that errors in the implementation can frequently be traced to errors in the specification. It has also been shown that specification errors are more likely than implementation errors to be safety critical [9].

Jaffe et al. have identified two properties of specifications that turn out to be particularly related to safety and accidents [13]: *Completeness* which ensures that there is a specified behavior for every input; and *consistency* which ensures that the system is free from conflicting requirements and undesired non-determinism.

Checking consistency and completeness of a realistic specification is nontrivial. Manual checking is tedious, error-prone, and infeasible for large specifications. Recently, automatic checking based on binary decision diagrams (BDDs) has been partially successful. However, since BDDs are used as a decision procedure for propositional logic, they can not

reason about arithmetic and inequalities, as is often required when analyzing software that deals with physical systems.

Many of the inequalities in TCAS II can be handled using BDDs by replacing the inequalities with Boolean variables. However, a number of the inequalities need to be interpreted and will produce spurious error reports when checked using BDDs. For instance, using BDDs it is not possible to determine that $x < 0 \wedge x > 600$ is false for some variable x. Heimdahl et al. observe that manually checking each error report is very labor intensive [9] and infeasible for large specifications. Anderson et al. note the inability of SMV to deal efficiently with the multiplication found in TCAS II [1].

Hoover and Chen [12] use a generalization of BDDs called *finite decision diagrams* (FDDs) to check consistency and completeness in the Tablewise tool. In contrast to BDDs with their binary branching, FDDs allow finite branching to represent finite valued variables. However, this is not sufficient to deal with the cases illustrated by the simple example above.

Another approach is to use a more general theorem-proving system, such as PVS [17]. Initial experiments checking properties of TCAS II were partly unsuccessful due to inefficiencies in an earlier version of PVS [8]. In general, the decision procedures are embedded within the prover and are therefore hard to access from an external specification checker. Often, they can only be accessed by generating, parsing, and type checking specifications in the theorem prover. This introduces significant overhead when checking a large number of properties. Furthermore, integrating and customizing a theorem prover in a special purpose specification checking tool is hard, due to the large amount of front-end proof-management software. Finally, although an unprovable theorem prover subgoal can implicitly provide some hints as to why a property is not true, it is not able to provide the user with localizing and debugging information in the form of an error trace.

The challenge is to find the right combination of deductive power (how rich a logic and what theories are decided) and efficiency (completion of verification in reasonable time). Unfortunately, deductive power and efficiency are usually inversely related. With too little deductive power, a tool may generate an inordinate number of spurious error reports since the proof procedure can not sufficiently interpret operators in the formulae. With too much deductive power, the verification may not terminate in an acceptable time period, since manual interaction is required in carrying out the verification.

In addition, it is important that counter-examples be presented to the user in the context of the original problem. In our experience, interpreting counter-example output and casting it in terms of the specification property to be verified is time consuming.

As an initial approach to solving these problems, we adapted the Stanford Validity Checker (SVC) [3], a general-purpose decision procedure for quantifier-free first-order logic with linear arithmetic, to check the consistency of specifications written in Requirements State Machine Language (RSML). We modified SVC to produce a counter-example in terms of the original specification, which enables a user to locate the corresponding flaw in the original specification when an inconsistency is discovered.

SVC's capabilities come closest to those of the General Validity Checker (GVC) [2], which is used to check consistency and completeness properties of Software Cost Reduction (SCR) tabular notations [10]. In contrast to SVC with its tight integration of decidable theories in a monolithic tool, GVC combines a collection of third party tools for propositional rewriting, tautology checking, and full Presburger arithmetic. Given a property, simple propositional rewriting is first tried. If it does not succeed, tautology checking, and later a Presburger arithmetic procedure is applied if needed. SVC does not support general quantification found in full Presburger arithmetic, but allows uninterpreted function symbols and supports *congruence closure*. That is, if $a = b$ then $f(a) = f(b)$. More experience is needed to determine the usefulness of this feature for checking safety critical specifications. On the other hand, it is our experience that unlimited quantification is not common in safety critical specifications and support for full Presburger arithmetic is therefore not needed.

We used SVC to check the consistency of a part of version 6.04a of the TCAS II specification, written in RSML. Our efforts uncovered an inconsistency. We confirmed with the maintainers of the specification that this inconsistency was a true problem in the specification. It was found independently by them and fixed as part of a general revision in version 7.0 of the TCAS II specification [7]. This correction was verified by reapplying our tools to the new specification.

The following sections describe the tools and their application to TCAS II. The Stanford Validity Checker is described in Section 2 and Section 3 describes RSML. An overview of TCAS II is given in Section 4 and the approach to verification is illustrated in Section 5. A discussion is provided in Section 6.

2 Stanford Validity Checker

The Stanford Validity Checker (SVC) [3] is a decision procedure for quantifier-free first-order logic and uses an algorithm similar to the algorithms by Shostak [20, 19] and Nelson-Oppen [16].

The TCAS relevant logic that SVC decides includes:

- the usual *Boolean connectives*;
- *equality* on terms;
- *uninterpreted function symbols*, such as: f, g, for which no previous knowledge is asserted;
- *congruence closure*, so that if $x = y$, then we know that $f(x) = f(y)$;
- *distinct constants*, which include the Boolean truth values, the rational numbers, and a form of enumerated constants (preceded with @). As an example, the distinct constant *@On_Ground* is not equal to any other constant;
- *linear arithmetic*, with the rational numbers, the usual inequalities: $<$, $>$, $\geq$, $\leq$, and expressions formed by addition and linear multiplication (i.e., constants multiplied by variables): $1/2 * x + 3/4 * y$; and
- *conditionals* in the form of if-then-else expressions, e.g., *if a then 1 else 2*,

and combinations of these. Furthermore, SVC supports interpreted theories such as *records* and (infinite) *stores*. It also supports *bitvectors*, which have been shown to be useful in hardware verification. SVC is modular in its implementation and facilitates easy extensions of new interpreted theories.

Given a formula, SVC will decide the validity of the formula and either return with "OK" or a counter-example. A

Transition(s): Potential-Threat ⟶ Other-Traffic

Location: Other-Aircraft ▷ Intruder-Status$_{s\text{-}154}$
Trigger Event: Air-Status-Evaluated-Event$_{e\text{-}396}$
Condition:

						OR						
A N D	Alt-Reporting$_{s\text{-}148}$ **in state** Lost	T	T	·	·	T	T	·	·	·	·
	RA-Mode-Cancelled$_{m\text{-}161}$	·	·	T	T	·	·	T	T	·	·
	Alt-Reporting$_{s\text{-}148}$ **in state** No	·	·	T	T	·	·	T	T	·	·
	Other-Bearing-Valid$_{v\text{-}133}$ = True	F	·	F	·	F	·	F	·	·	·
	Other-Range-Valid$_{v\text{-}130}$ = True	·	F	·	F	·	F	·	F	·	·
	Potential-Threat-Range-Test$_{m\text{-}289}$	T	T	T	T	F	F	F	F	·	·
	Potential-Threat-Condition$_{m\text{-}288}$	·	·	·	·	·	·	·	·	F	·
	Proximate-Traffic-Condition$_{m\text{-}292}$	·	·	·	·	T	T	T	T	F	·
	Threat-Condition$_{m\text{-}305}$	·	·	·	·	·	·	·	·	F	·
	PT-Timer$_{s\text{-}240}$ **in state** 0	·	·	·	·	T	T	T	T	T	·
	Other-Air-Status$_{s\text{-}148}$ **in state** On-Ground	·	·	·	·	·	·	·	·	·	T

Output Action: Intruder-Status-Evaluated-Event$_{e\text{-}396}$

Figure 1: A transition definition from TCAS II with the guarding condition expressed as an AND/OR table. T denotes logical *true* and F denotes *false*.

counter-example is a conjunction of simple predicates which is (1) satisfiable and (2) implies the negation of the original formula.

SVC is implemented in C++ and care has been taken to make it efficient. It has been applied to a number of large hardware verification examples, including verification of FLASH [21].

3 Requirements State Machine Language (RSML)

RSML was developed as a requirements specification language for embedded systems. The language is based on hierarchical finite state machines and is similar to David Harel's Statecharts [5, 6]. As illustrated in Figure 2, RSML supports parallelism, hierarchies, and guarded transitions which originated in Statecharts. Transition guards are expressed as *e[c]/a* where *e* is the trigger event that enables a transition, *c* is a guarding condition that determines under which conditions the transition can be taken, and *a* is the set (possibly empty) of events that are generated as actions when the transition is taken.

In real-life specifications such as TCAS II, the guarding conditions required to accurately capture the requirements are often complex. The propositional logic notation traditionally used to define these conditions do not scale well to complex expressions and quickly become unreadable. To overcome this problem, guarding conditions are expressed in a tabular representation of disjunctive normal form (DNF) called AND/OR tables (see Figure 1 for an example from the TCAS II requirements). The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements are true. A dot denotes "don't care." The table evaluates to false if all of the columns are false.

To further improve readability, many other syntactic conventions were introduced in RSML. For example, expressions used in the predicates can be defined as mathematical functions (e.g., Other-Tracked-Alt$_{f\text{-}343}$), and familiar

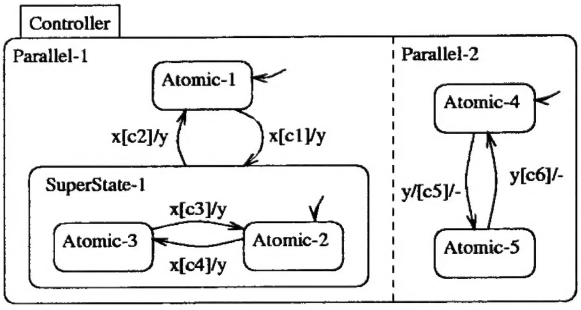

Figure 2: An example of a hierarchical state machine.

and frequently used conditions can be defined as macros (e.g., Threat-Condition$_{m\text{-}305}$). A macro is simply a named AND/OR table defined elsewhere in the document. The subscript is used to indicate the type of an identifier (f for functions, m for macros, and v for variables) and gives the page number in the TCAS II requirements document where the identifier is defined. Naturally, the state machine in a real system is seldom as simple as in Figure 2. As an example of a realistic model, a part of the state machine modeling an intruding aircraft in TCAS II is shown in Figure 3. See [14] for a detailed description of the graphical notation.

RSML specifications are required to be deterministic and have a well-defined behavior for all inputs. A number of criteria must be satisfied by a specification to guarantee this [9]. In particular, two properties must be satisfied locally by transitions out of a state:

1. *Consistency:* Every pair of transitions out of the same state and with the same trigger event must have mutually exclusive guarding conditions; exactly one of these transitions can be taken at any time. This can be shown by pairwise conjuncting the guarding conditions on transitions triggered by the same event out of a state and show that the conjunction forms a contradiction;

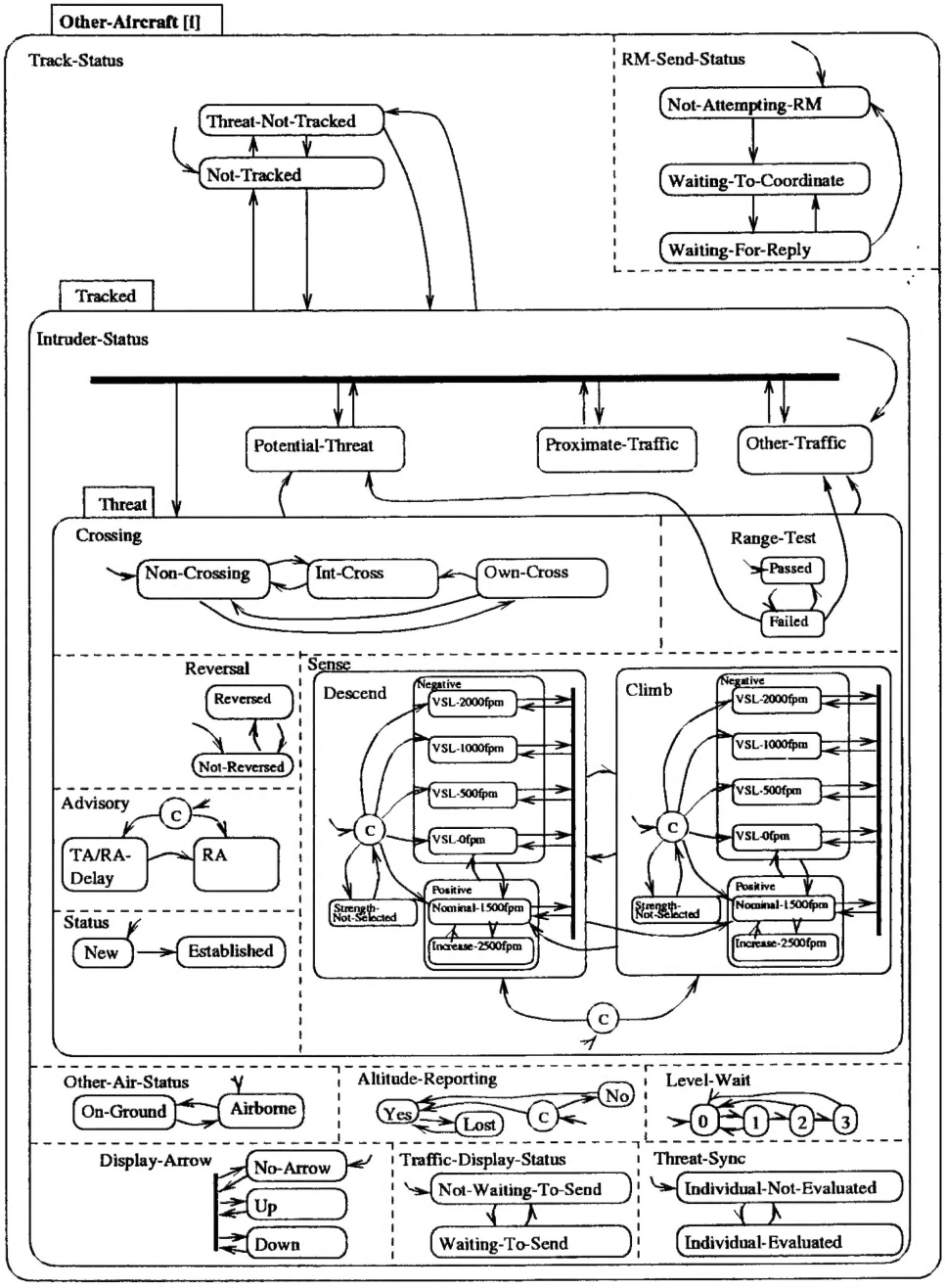

Figure 3: RSML model of an intruding aircraft. The guarding conditions are specified separately and are therefore not shown. The circles marked C denote conditional entry points. For instance, in state machine *Sense*, the condition specifies the selection of *Climb* or *Descend*.

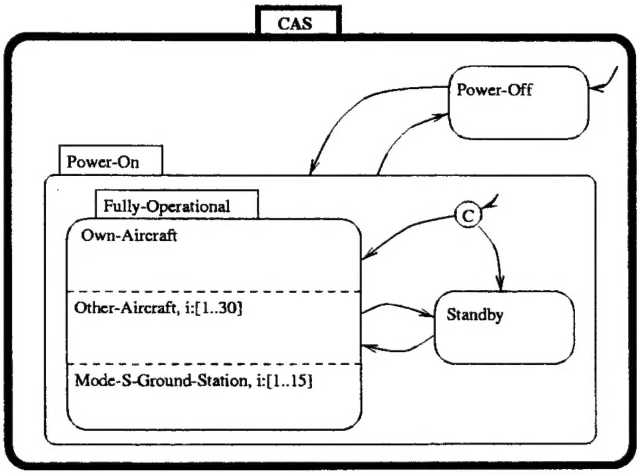

Figure 4: Collision Avoidance Subsystem (highest level).

2. *Completeness:* The disjunction of the guarding conditions must form a tautology. This ensures that there is always a transition to take.

It was not possible to check completeness of the guarding conditions in TCAS II, since the creators of the TCAS II specification implicitly assume a self-loop in the specification when no guarding condition is satisfied (i.e., if no transition is enabled in a state, the automaton remains in that state). The work described in this paper has therefore focused on the task of verifying consistency.

4 Testbed Specification

To introduce the reader to our case study, we provide a short overview of TCAS II.

4.1 TCAS II

TCAS is a family of airborne devices that function independently of the ground-based air traffic control (ATC) system to provide collision avoidance protection for commercial aircraft and larger aircraft. TCAS II provides traffic advisories and recommended escape maneuvers (resolution advisories) in a vertical direction to avoid aircraft collisions.

We have focussed on the part of the Collision Avoidance Subsystem (CAS) in TCAS II that classifies intruding aircraft as *Other-Traffic*, *Proximate-Traffic*, *Potential-Threat*, or *Threat*.

CAS: The highest level CAS state machine is shown in Figure 4. At this level, CAS is either on or off; if it is on, it may be either fully operational or in standby mode.

In the CAS logic, the states of three components are modeled: own aircraft, other aircraft, and ground radar stations. We focus on the specification of other aircraft.

Other-Aircraft: The state machines defining how an intruding aircraft is modeled in TCAS II can be seen in Figure 3. The guarding conditions are specified separately. In short, the top-level state machine reflects whether a particular Other-Aircraft is currently being tracked or not.

The *Intruder-Status* state within Tracked reflects the current classification of *Other-Aircraft* (*Other-Traffic*, *Proximate-Traffic*, *Potential-Threat*, and *Threat*). When an intruder is classified as a threat, a two-step process is used to select a Resolution Advisory (RA). The first step is to select a sense (Climb or Descend). Based on the range and altitude tracks of the intruder, the CAS logic models the intruder's path until Closest Point of Approach (CPA). The CAS logic computes the predicted vertical separation for both climb and descend maneuvers, and selects the sense that provides the greater vertical separation.

The second step in selecting a RA is to select the strength of the advisory. The least disruptive vertical rate maneuver that will still achieve safe separation is selected. For a more complete description of TCAS II and how it was modeled using RSML, the reader is referred to [14].

4.2 Range Test

Based on experience from the definition of the TCAS II specification, we decided to focus our effort on the transitions between the states *Threat*, *Potential-Threat*, *Proximate-Traffic*, and *Other-Traffic*. The guarding conditions of these transitions involve some of the most complex interactions and include a large number of inequalities and two non-linear expressions. To illustrate, the guarding conditions of the transitions from *Proximate-Traffic* to *Potential-Threat* and from *Proximate-Traffic* to *Other-Traffic* and the relevant macros are shown in Appendix C. One of these macros is the macro *Potential-Threat-Range-Test* as shown in Figure 5. Note that it has a non-linear expression in the first row and a number of linear inequalities. The expressions *H1TA*, *DMODTA*, and *TRTHRTA* are look-up tables that return a constant value depending on the system state. The abbreviation *TAURTA* is a function that for some cases returns an expression which is non-linear.

5 Verifying Properties of TCAS II

5.1 Approach

We focused on checking the transitions out of the *Threat*, *Potential-Threat*, *Proximate-Traffic*, and *Other-Traffic* states, as shown in Figure 3. For the *Threat* state we checked the consistency between pairs of transitions going out of the *Passed* and *Failed* states. From *Failed*, transitions are possible to *Passed*, *Potential-Threat*, and *Other-Traffic*. From *Passed*, transitions are possible to *Failed*, *Potential-Threat*, and *Other-Traffic*. The transitions to the latter two are indicated by the transitions from the boundary of *Threat*.

In our initial experiments, we used a textual version of RSML and an RSML to PVS translator that Czerny et al. constructed in a previous effort [8]. The proof obligation was expanded out in PVS and passed on to SVC for verification, using a translator from PVS to SVC previously constructed at Stanford. The counter-example was interpreted manually in terms of the RSML definitions.

As shall be described below, the first discrepancies were quickly detected. However, the overall verification required manual intervention to direct the proof[1] and the proof obligations were relatively large and slow to check in PVS due to lack of structure sharing of formulae. Also, manually presenting the counter-examples in RSML format in the context of the TCAS II specification was very time consuming.

[1]This could be automated by writing an automated strategy in PVS.

Macro: Potential-Threat-Range-Test
Definition:

		OR	
AND	Other-Tracked-Range$_{f\text{-}347}$ * Other-Tracked-Range-Rate$_{f\text{-}348}$ $\leq$ H1TA	·	T
	Other-Tracked-Range-Rate$_{f\text{-}348}$ > 10 ft/s$_{(RDTHRTA)}$	F	T
	Other-Tracked-Range$_{f\text{-}347}$ $\leq$ DMODTA	·	T
	TAURTA < TRTHRTA	T	·

Figure 5: The macro *Potential-Threat-Range-Test.*

For the examples we tried, we were able to carry out the verification with two trivial substitutions for the expressions involving non-linear multiplication and division with uninterpreted functions. We therefore decided to use SVC directly in the verification and construct a prototype verification tool based on SVC.

5.2 The Prototype Tool

The prototype verification tool, Analyzer, mechanizes the process of expanding formulae, invoking SVC, and displaying the counter-example in a format that is understandable to the RSML engineer. The tool is written in Lisp and connects to SVC using a foreign function interface.

Verification using the tool is done as follows. Selected parts of the TCAS II specification, represented in an ASCII version of RSML (see Appendix A for an example), are parsed and a database of the transitions and definitions is created. Next, on the users request, two guarding conditions to be checked for consistency, E_1 and E_2, are expanded. This process involves recursively expanding all macro references in the guarding conditions. The consistency property is then generated as $\neg(E_1 \wedge E_2)$.

Of the two main types of abstractions in RSML, macros and functions, we chose to leave functions undefined, whereas we expanded all macros from the beginning. Furthermore, we substituted non-linear expressions with variables. This policy seems to work well. First, in the parts we looked at, the TCAS II specification is written so that it will be well-formed regardless of the details of function definitions. To illustrate, we let the function *TAURTA* remain unexpanded in the *Potential-Threat-Range-Test* macro shown in Figure 5, and we substituted the non-linear multiplication in the first line with a variable. Second, expanding all macro references did not in most cases significantly affect the complexity of the proof. However, expanding out all macros may in some cases result in large formulae that require a significant number of computations to verify. Fortunately, in some instances it may still be possible to establish consistency by selectively expanding some of the macro references and attempting the verification again. Our method can easily be extended to adopt such a scheme.

After the construction of the formula is completed, SVC is called to decide it. SVC may return "valid", indicating that no problems were found. More often, it returns a counter-example. At this point, Analyzer traces through the two transition tables and recursively dependent macro tables to determine which tables had to evaluate to true (or false) in the counter-example context in order for both transition tables to be true. In effect, Analyzer is determining the localized information that explains the counter-example in terms of RSML tables. The truth value of a table is determined by evaluating the predicates at the leftmost column of the table in the context of the counter-example. Predicates can evaluate to true, false, or no value (if the truth value of the predicate can not be determined in the counter-example context). The tables whose truth values are involved in the counter-example are then pretty printed along with the SVC counter-example translated into RSML terms.

As an illustration, consider the partial output of the tool from checking consistency between two transitions *Proximate-Traffic* to *Potential-Threat* and *Proximate-Traffic* to *Other-Traffic* as shown in Figure 6. The output first lists the conditions in the counter-example under which the guarding conditions of the transitions are simultaneously true. The relevant tables are then listed. For tables that are true (e.g., tables for the transitions), the column that is satisfied is indicated by an arrow directly underneath it. The counter-example assertions satisfying the column are specified to the right of the rows (an arrow indicates a macro reference). For tables that are false, the counter-example assertions falsifying each column are indicated in parenthesis to the right of the rows.

The first table in Figure 6 evaluates to true since the fourth column (disjunct) is true in the counter-example context. The column represents a conjunction of equalities and a macro `M_POTENTIAL_THREAT_RANGE_TEST`, all of which have to be false. Since `M_POTENTIAL_THREAT_RANGE_TEST` must evaluate to false, the macro is then listed. It is false since the first inequality is false and the second is true. This causes both disjuncts of the table to be false. The full error report is illustrated in Appendix B.

5.3 Results

Initially, almost all of the pairs of transitions that we tested produced counter-examples. It turned out that some of these were due to missing information regarding the global state. Two invariants were added, and the analyses were rerun.

Out of 19 pairs of transitions analyzed, we found 6 discrepancies. The discrepancies were given to the maintainers of TCAS II, who reported that 5 of these were due to a difference in interpretation of the semantics: in their view, transitions out of a super-state have higher priority than transitions that originate in its sub-states. However, the analysis of the pair of transitions shown in Appendix B revealed a non-determinism in the specification. This is the problem that was found independently by the maintainers of the TCAS II specification. We have verified the correction in version 7.0 using Analyzer.

Although the violation of consistency documented in Appendix B is relatively simple, the advantage of using a powerful decision procedure such as SVC is indirectly realized in

```
Transition1: M_PROXIMATE_TRAFFIC_TO_POTENTIAL_THREAT[]
Transition2: M_PROXIMATE_TRAFFIC_TO_OTHER_TRAFFIC[]

COUNTER-EXAMPLE

1) (S_ALT_REPORTING = @S_NO)
2) (NOT (S_PT_TIMER = @S_PT_0)
3) (NOT (PAIR1 <= F_H1TA[]))
4) (F_OTHER_TRACKED_RANGE_RATE[] > 10)
5) (NOT (V_OTHER_BEARING_VALID = @BOOL_CTRUE))
6) (NOT (V_PREV_RA_INHIBIT = @BOOL_CTRUE))
7) (S_AUTO_SL = @S_ASL_2)


*** TABLE: M_PROXIMATE_TRAFFIC_TO_POTENTIAL_THREAT[] / TRUE ***

  (S_ALT_REPORTING = @S_YES)                F T F F T ;1
  (V_OTHER_BEARING_VALID = @BOOL_CTRUE)     T * T * * ;
  (V_OTHER_RANGE_VALID = @BOOL_CTRUE)       T * T * * ;
  M_POTENTIAL_THREAT_CONDITION[]            T T * * * ;
  M_THREAT_CONDITION[]                      * F * * F ;
  (S_PT_TIMER = @S_PT_0)                    * * F F F ;2
  M_POTENTIAL_THREAT_RANGE_TEST[]           * * * F * ;<-
                                                  ^


*** TABLE: M_PROXIMATE_TRAFFIC_TO_OTHER_TRAFFIC[] / TRUE ***

  (S_ALT_REPORTING = @S_LOST)                    T T * * * * * * ;
  M_RA_MODE_CANCELLED[]                          * * T T * * * * ;<-
  (S_ALT_REPORTING = @S_NO)                      * * T T T T * * ;1
  (V_OTHER_BEARING_VALID = @BOOL_CTRUE)          F * F * F * * * ;5
  (V_OTHER_RANGE_VALID = @BOOL_CTRUE)            * F * F * F * * ;
  M_PROXIMATE_TRAFFIC_CONDITION[]                * * * * * * F * ;
  M_POTENTIAL_THREAT_RANGE_TEST[]                * * * * T T * * ;
  M_POTENTIAL_THREAT_CONDITION[]                 * * * * * * F * ;
  M_THREAT_CONDITION[]                           * * * * * * F * ;
  (S_OTHER_AIR_STATUS = @S_STATE_ON_GROUND)      * * * * * * * T ;
                                                       ^


*** TABLE: M_POTENTIAL_THREAT_RANGE_TEST[] / FALSE ***

  (PAIR1 <= F_H1TA[])                            * T ;(3)
  (F_OTHER_TRACKED_RANGE_RATE[] > 10)            F T ;(4)
  (F_OTHER_TRACKED_RANGE[] <= F_DMODTA[])        * T ;
  (F_TAURTA[] < F_TRTHRTA[])                     T * ;

[...]

----
* PAIR1 = F_OTHER_TRACKED_RANGE[] * F_OTHER_TRACKED_RANGE_RATE[]
```

Figure 6: Counter-example. PAIR1 is the variable that has been substituted for the non-linear expression. The numbers next to the macros denote assertion numbers.

the output of fewer spurious counter-examples. As an experiment, we left the inequalities uninterpreted in the consistency checks between the transitions *Other-Traffic* to *Threat* and *Other-Traffic* to *Proximate-Traffic*. The resulting spurious counter-example required both

$$\textit{Own-Tracked-Alt-Rate} > 600$$

and

$$\textit{Own-Tracked-Alt-Rate} < 0$$

to be true at the same time. This and other false negatives similar to it were not generated when using the full capability of SVC.

Having decision procedures for linear arithmetic also avoids spurious error reports caused by using syntactically different but semantically equivalent formulae in the specification. As a simple example, a state machine in TCAS II models whether or not an aircraft is descend inhibited; an aircraft too close to the ground is not allowed to descend. The guarding conditions in this state machine contain the predicates

$$\textit{Own-Tracked-Alt} - \textit{Ground-Level} > 1200$$

and

$$\textit{Own-Tracked-Alt} \leq 1200 + \textit{Ground-Level}.$$

An analysis method without decision procedures for linear arithmetic would either fail to detect that these two predicates are contradictory and generate spurious error reports, or require formulae to be in a standard form.

To illustrate typical run-times of the Analyzer prototype tool, Figure 7 lists the run-times required to calculate the first counter-example for the pairs of transitions out of *Proximate-Traffic*. In general, the time required to find a counter-example using the prototype tool was less than 1 second of runtime on a Linux based Intel Pentium Pro machine and the time required to pretty print took at most 38 seconds. In comparison, expanding each proof obligation in PVS and getting a counter-example in SVC and PVS took roughly 5-10 minutes. Writing up the counter-examples as a list of macros took more than 30 minutes for each discrepancy. Overall, using SVC and the pretty printer tool decreased the time to get a counter-example by several orders of magnitude.

The time required by SVC to finally get a "valid" after invariants were added was in most cases less than 5 minutes. In one case, SVC required approximately 20 minutes. We expect that these numbers could be significantly lowered by applying a number of extra simplification heuristics in SVC. Alternatively, only expanding certain selected macros could possibly lower these numbers as well.

The Analyzer prototype tool was built in Lisp in less than a man month by a student who was unfamiliar with SVC and Lisp. This was possible since we were able to call the SVC proof engine directly from Lisp. All computations of truth values of RSML formulae were done in SVC.

6 Discussion

The class of formulae that SVC decides (quantifier free first-order logic with linear arithmetic) was sufficient for the part of TCAS II that we checked. Other parts of TCAS II contain non-linear arithmetic and quantification in the guarding conditions and therefore can not directly be decided by the current version of SVC. We plan to extend SVC with new theories to handle a larger class of formulae. In the long-term, we can not expect to fully automate the verification process. A better strategy would be to do as much automatically as possible, but use a general-purpose theorem prover to handle constructs that can not be dispatched automatically. We plan to use PVS for this purpose.

Checking local properties of specifications may require global information. In our experiments, two invariants were needed in order to discharge the consistency proof obligations. Methods for extracting such relevant information from the surrounding system are crucial for application of local checks on a larger scale. To address this issue, efficient invariant generation techniques are being developed by the authors to detect global invariants in RSML.

Using SVC's open interface, we were able to quickly build the Analyzer prototype on top of SVC, exploiting as much functionality of SVC as possible. In our opinion, open and highly efficient verification tools are essential for successful application of formal methods to the verification of safety critical systems.

Transition		1st Counter-Example	
from State	to States	SVC (msec)	Total (sec)
Proximate-Traffic	Other-Traffic/Threat	10	30.4
Proximate-Traffic	Potential-Threat/Threat	20	37.7
Proximate-Traffic	Other-Traffic/Pot. Threat	10	24.5

Figure 7: Examples of run-times on an Intel Pentium Pro based machine running Linux. The table shows the pairs of transitions being checked, the CPU time required for SVC to calculate the first counter-example, and the total CPU time to calculate and display the first counter-example.

We have reported on an initial feasibility study of checking consistency properties of RSML specifications. More effort is needed to investigate the application of formal verification tools to check other aspects of RSML specifications. The long term goal is to construct an integrated RSML toolset. The SCR toolset [11] is inspirational in this regard.

Acknowledgements

We would like to thank John Rushby, SRI International, who made the contact between the research groups and provided valuable comments during the project and to this paper. We also appreciate the comments from the anonymous reviewers and Nancy Leveson, University of Washington, to earlier versions of the paper. Finally, we would like to thank Ramesh Bharadwaj and Constance Heitmeyer, Naval Research Laboratory, for insightful comments on the SCR tools, and Jeff Thompson, University of Minnesota, who extended an existing RSML parser to enable translations of RSML specifications into Analyzer input format.

References

[1] R.J. Anderson, P.Beame, S. Burns, W. Chan, F. Modugno, Notkin D, and J.D. Reese. Model checking large software specifications. In D. Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96)*, pages 156–166, October 1996.

[2] R. Bharadwaj. A generalized validity checker. Technical Report VALID/96, Version 1.0, Software Engineering Section, Naval Research Laboratory, June 1996. Research Note.

[3] C. Barrett D.L. Dill and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design (FMCAD)*, number 1166 in Lecture Notes in Computer Science, pages 197–201. Springer-Verlag, November 1996.

[4] S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[7] M. Heimdahl and M. Rubinstein. Private communication between Mats Heimdahl and Mike Rubinstein, Rannoch, July 1997.

[8] M. P.E. Heimdahl and B.J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, 1996.

[9] M. P.E. Heimdahl and N.G. Leveson. Completeness and consistency analysis of state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[10] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *TOSEM*, 5(3):231–261, July 1996.

[11] C. L. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of 12th Annual Conference on Computer Assurance (COMPASS '97)*, pages 35 – 47, Gaithersburg, MD, USA, June 1997.

[12] D.N. Hoover and Zewei Chen. Tablewise, a decision table tool. In J. Rushby, editor, *Proceedings of 10th Annual Conference on Computer Assurance (COMPASS '95)*, pages 97–108, Gaithersburg, MD, USA, June 1995. IEEE.

[13] M. S. Jaffe, N. G. Leveson, M. P.E. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[14] N.G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J.D. Reese. Completeness and consistency analysis of state-based requirements. *IEEE Transactions on Software Engineering*, 20(9):694–707, September 1994.

[15] Michael J. Morgan. Integrated modular avionics for next-generation commercial airplanes. *IEEE Aerospace and Electronic Systems Magazine*, 6(8):9–12, August 1991.

[16] G.E. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[17] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[18] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.

[19] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[20] R.E. Shostak. Deciding combinations of theories. Technical Report SRI-CSL-132, Computer Science Laboratory, SRI International, February 1982.

[21] Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic generation of invariants in processor verification. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design (FMCAD)*, number 1166 in Lecture Notes in Computer Science, pages 197–201. Springer-Verlag, November 1996.

A Analyzer Input

```
TRANSITION Proximate_Traffic TO Potential_Threat :
  LOCATION : TCAS_Controller_Power_On_Fully_Operational_\
     Other_Aircraft_Track_Status_Tracked_Intruder_Status
  TRIGGER : Air_Status_Evaluated_Event
  CONDITION : TABLE
    Alt_Reporting IN_STATE Yes      : F T F F T ;
    Other_Bearing_Valid = cTrue     : T . T . . ;
    Other_Range_Valid = cTrue       : T . T . . ;
    Potential_Threat_Condition()    : T T . . . ;
    Threat_Condition()              : . F . . F ;
    PT_Timer IN_STATE PT_0          : . . F F F ;
    Potential_Threat_Range_Test()   : . . . F . ;
    END TABLE
  ACTION : Intruder_Status_Evaluated_Event
END TRANSITION


TRANSITION Proximate_Traffic TO Other_Traffic :
LOCATION : TCAS_Controller_Power_On_Fully_Operational_\
           Other_Aircraft_Track_Status_Tracked_Intruder_Status
 TRIGGER : Air_Status_Evaluated_Event
 CONDITION : TABLE
   Alt_Reporting IN_STATE Lost                : T T . . . . . . ;
   RA_Mode_Cancelled()                        : . . T T . . . . ;
   Alt_Reporting IN_STATE No                  : . . T T T T . . ;
   Other_Bearing_Valid = cTrue                : F . F . F . . . ;
   Other_Range_Valid = cTrue                  : . F . F . F . . ;
   Proximate_Traffic_Condition()              : . . . . . . . F ;
   Potential_Threat_Range_Test()              : . . . . . T T . ;
   Potential_Threat_Condition()               : . . . . . . . F ;
   Threat_Condition()                         : . . . . . . . F ;
   Other_Air_Status IN_STATE State_On_Ground  : . . . . . . . T ;
   END TABLE
 ACTION : Intruder_Status_Evaluated_Event
END TRANSITION


MACRO Potential_Threat_Range_Test() :
TABLE
  (Other_Tracked_Range() *
          Other_Tracked_Range_Rate()) <= H1TA() : . T ;
  Other_Tracked_Range_Rate() > RDTHRTA          : F T ;
  Other_Tracked_Range() <= DMODTA()             : . T ;
  TAURTA() < TRTHRTA()                          : T . ;
END TABLE
END MACRO


MACRO RA_Mode_Cancelled() :
TABLE
  RA_Inhibit()           : T ;
  PREV_RA_Inhibit = cTrue : F ;
END TABLE
END MACRO


MACRO RA_Inhibit() :
TABLE
  Auto_SL IN_STATE ASL_2          : T . . ;
  Mode_Selector = TA_Only         : . T . ;
  RA_Inhibit_From_Ground = cTrue  : . . T ;
END TABLE
END MACRO
```

B Counter-Example

```
<<<<<<<<<<<<<<<<<<<< ANALYZER >>>>>>>>>>>>>>>>>>>>

0) Quit
1) List all tables
2) View a table
3) Check the validity of an expression
4) Check consistency between transitions

Your choice: 4

Transition1: M_PROXIMATE_TRAFFIC_TO_POTENTIAL_THREAT[]
Transition2: M_PROXIMATE_TRAFFIC_TO_OTHER_TRAFFIC[]

COUNTER-EXAMPLE

1) (S_ALT_REPORTING = @S_NO)
2) (NOT (S_PT_TIMER = @S_PT_0)
3) (NOT (PAIR1 <= F_H1TA[]))
4) (F_OTHER_TRACKED_RANGE_RATE[] > 10)
5) (NOT (V_OTHER_BEARING_VALID = @BOOL_CTRUE))
6) (NOT (V_PREV_RA_INHIBIT = @BOOL_CTRUE))
7) (S_AUTO_SL = @S_ASL_2)


*** TABLE: M_PROXIMATE_TRAFFIC_TO_POTENTIAL_THREAT[] / TRUE ***

  (S_ALT_REPORTING = @S_YES)              F T F F T ;1
  (V_OTHER_BEARING_VALID = @BOOL_CTRUE)   T * T * * ;
  (V_OTHER_RANGE_VALID = @BOOL_CTRUE)     T * T * * ;
  M_POTENTIAL_THREAT_CONDITION[]          T T * * * ;
  M_THREAT_CONDITION[]                    * F * * F ;
  (S_PT_TIMER = @S_PT_0)                  * * F F F ;2
  M_POTENTIAL_THREAT_RANGE_TEST[]         * * * F * ;<-
                                                ^

*** TABLE: M_PROXIMATE_TRAFFIC_TO_OTHER_TRAFFIC[] / TRUE ***

  (S_ALT_REPORTING = @S_LOST)               T T * * * * * * ;
  M_RA_MODE_CANCELLED[]                     * * T T * * * * ;<-
  (S_ALT_REPORTING = @S_NO)                 * * T T T T * * ;1
  (V_OTHER_BEARING_VALID = @BOOL_CTRUE)     F * F * F * * * ;5
  (V_OTHER_RANGE_VALID = @BOOL_CTRUE)       * F * F * F * * ;
  M_PROXIMATE_TRAFFIC_CONDITION[]           * * * * * * F * ;
  M_POTENTIAL_THREAT_RANGE_TEST[]           * * * * T T * * ;
  M_POTENTIAL_THREAT_CONDITION[]            * * * * * * F * ;
  M_THREAT_CONDITION[]                      * * * * * * F * ;
  (S_OTHER_AIR_STATUS = @S_STATE_ON_GROUND)  * * * * * * * T ;
                                                      ^

*** TABLE: M_POTENTIAL_THREAT_RANGE_TEST[] / FALSE ***

  (PAIR1 <= F_H1TA[])                       * T ;(3)
  (F_OTHER_TRACKED_RANGE_RATE[] > 10)       F T ;(4)
  (F_OTHER_TRACKED_RANGE[] <= F_DMODTA[])   * T ;
  (F_TAURTA[] < F_TRTHRTA[])                T * ;


*** TABLE: M_RA_MODE_CANCELLED[] / TRUE ***

  M_RA_INHIBIT[]                      T ;<-
  (V_PREV_RA_INHIBIT = @BOOL_CTRUE)   F ;6
                                      ^

*** TABLE: M_RA_INHIBIT[] / TRUE ***

  (S_AUTO_SL = @S_ASL_2)                      T * * ;7
  (V_MODE_SELECTOR = @MODE_TA_ONLY)           * T * ;
  (V_RA_INHIBIT_FROM_GROUND = @BOOL_CTRUE)    * * T ;
                                              ^

----
* PAIR1 = F_OTHER_TRACKED_RANGE[] * F_OTHER_TRACKED_RANGE_RATE[]
** Two global invariants rule out the previous counter-examples.
```

C RSML Example

Transition(s): Proximate-Traffic ⟶ Potential-Threat

Location: Other-Aircraft ▷ Intruder-Status$_{s\text{-}154}$

Trigger Event: Air-Status-Evaluated-Event$_{e\text{-}396}$
Condition:

				OR			
A N D	Alt-Reporting$_{s\text{-}148}$ **in state** Yes	F	T	F	F	T	
	Other-Bearing-Valid$_{v\text{-}133}$ = True	T	·	T	·	·	
	Other-Range-Valid$_{v\text{-}130}$ = True	T	·	T	·	·	
	Potential-Threat-Condition$_{m\text{-}288}$	T	T	·	·	·	
	Threat-Condition$_{m\text{-}305}$	·	F	·	·	F	
	PT-Timer$_{s\text{-}240}$ **in state** 0	·	·	F	F	F	
	Potential-Threat-Range-Test$_{m\text{-}289}$	·	·	·	F	·	

Output Action: Intruder-Status-Evaluated-Event$_{e\text{-}396}$

Transition(s): Proximate-Traffic ⟶ Other-Traffic

Location: Other-Aircraft ▷ Intruder-Status$_{s\text{-}154}$

Trigger Event: Air-Status-Evaluated-Event$_{e\text{-}396}$
Condition:

						OR				
A N D	Alt-Reporting$_{s\text{-}148}$ **in state** Lost	T	T	·	·	·	·	·	·	
	RA-Mode-Cancelled$_{m\text{-}161}$	·	·	T	T	·	·	·	·	
	Alt-Reporting$_{s\text{-}148}$ **in state** No	·	·	T	T	T	T	·	·	
	Other-Bearing-Valid$_{v\text{-}133}$ = True	F	·	F	·	F	·	·	·	
	Other-Range-Valid$_{v\text{-}130}$ = True	·	F	·	F	·	F	·	·	
	Proximate-Traffic-Condition$_{m\text{-}292}$	·	·	·	·	·	·	F	·	
	Potential-Threat-Range-Test$_{m\text{-}289}$	·	·	·	·	T	T	·	·	
	Potential-Threat-Condition$_{m\text{-}288}$	·	·	·	·	·	·	F	·	
	Threat-Condition$_{m\text{-}305}$	·	·	·	·	·	·	F	·	
	Other-Air-Status$_{s\text{-}148}$ **in state** On-Ground	·	·	·	·	·	·	·	T	

Output Action: Intruder-Status-Evaluated-Event$_{e\text{-}396}$

Macro: Potential-Threat-Range-Test
Definition:

		OR	
A N D	Other-Tracked-Range$_{f\text{-}347}$ * Other-Tracked-Range-Rate$_{f\text{-}348}$ ≤ H1TA	·	T
	Other-Tracked-Range-Rate$_{f\text{-}348}$ > 10 ft/s$_{(RDTHRTA)}$	F	T
	Other-Tracked-Range$_{f\text{-}347}$ ≤ DMODTA	·	T
	TAURTA < TRTHRTA	T	·

Macro: RA-Mode-Cancelled
Definition:

A N D	RA-Inhibit$_{m\text{-}293}$	T
	PREV(RA-Inhibit$_{m\text{-}293}$)	F

Macro: RA-Inhibit
Definition:

			OR	
A N D	Auto-SL$_{s\text{-}31}$ **in state** 2	T	·	·
	Mode-Selector$_{v\text{-}36}$ = TA_Only	·	T	·
	RA-Inhibit-From-Ground$_{m\text{-}293}$	·	·	T